HINT: Hierarchical INference with Task-demonstrations

Nirlipta Pande [1] **Georg Niess** [1] **Julian Gutheil** [1] **Roman Kern** [1] **Robert Legenstein** [1] **Robert Peharz** [1]

Abstract

The Abstract Reasoning Corpus (ARC) evaluates abstract reasoning through tasks that are easy for humans yet remain challenging for AI systems. Recent inductive approaches treat each task in isolation and struggle to acquire new skills efficiently. We identify two key methodological gaps responsible: 1) *experience gap* where treating each task independently discards prior experience, forcing models to rediscover previously seen compositional concepts and 2) *reasoning gap* which means that without prior knowledge about possible solutions, the search space becomes too large, causing reasoning to diverge and making correct solutions expensive to find. We address both by introducing HINT (Hierarchical INference with Task-demonstrations), a multi-phase framework with three key components: a library of previously solved ARC tasks; in-context natural-language reasoning guided by a small set of similar tasks from the library; and program synthesis conditioned on inductive biases from the previous phases. HINT achieves 20.8% accuracy at $0.28 per task on ARC-AGI-2 public evaluation dataset, compared to 5% for direct synthesis with the same backbone model without model finetuning or additional datasets. These results provide evidence that principled reuse of prior experience and hierarchical reasoning are necessary for efficient abstract reasoning.[1]

1. Introduction

Intelligence is not just the ability to solve a task, but to do so with limited prior knowledge and compute by acquiring and using relevant skills. The ARC-AGI benchmark tests a system's ability to induce abstract, compositional rules from sparse examples within a limited compute budget, also known as skill acquisition efficiency (ARC Prize Foundation, 2025; Chollet, 2019). ARC tasks are presented through a visual interface of color-coded grids, yet their underlying semantics are object-centric, relational, and compositional. While humans solve these tasks with near-zero error, under a limited compute budget of $1 per task, state-of-the-art vision language models attain at most 55.8% accuracy at a cost of 0.68$ per task on the ARC-AGI-2 validation dataset [2] (ARC Prize Foundation, 2025).

Humans perform well on ARC in part because they can draw on rich prior knowledge (Spelke & Kinzler, 2007) about two-dimensional objects, their attributes, and the transformations and relations that govern them—abstract concepts grounded in everyday experience with the physical world. When solving an ARC task, humans typically reason at a high level, drawing on their rich prior experience about the underlying concepts before translating this understanding into a concrete sequence of operations. In contrast, most existing AI approaches treat each task in isolation and attempt to directly map the transformation to a program or produce an output, often without an explicit intermediate abstraction (Greenblatt, 2024; Berman, 2024; 2025). Several works address this gap by performing expensive model finetuning, trading efficiency for accuracy (Pourcel et al., 2025; Li et al., 2025).

We hypothesize that this behavior results from two specific methodological gaps in existing inductive approaches. First, because the current work treats each ARC task in isolation, every hypothesis is formed from scratch within an unbounded transformation space. Without learning from previously attempted similar transformations, the models attempt an unconstrained generation and produce plausible-sounding but unfaithful reasoning chains that systematically fail to converge on the correct rule (Turpin et al., 2023). We call this the *experience gap*. Pang (2025) partially addresses this by maintaining a solution library and retrieving relevant programs to guide program synthesis. Second, the closest existing reasoning approach is hypothesis search (Wang et al., 2024b) which has two phases: first generating a natural-language description of the transformation rule, followed by program synthesis. While this outperforms direct

[1]Institute of Machine Learning and Neural Computation, Graz University of Technology, Graz, Austria. Correspondence to: Nirlipta Pande <nirliptapande@gmail.com>.

Accepted to the 2nd Workshop on Compositional Learning at ICML 2026, Seoul, South Korea. Copyright 2026 by the author(s).

[1]Code available here

[2]Model comparisons are based on systems available as of February 15, 2026, corresponding to the completion of our experiments.

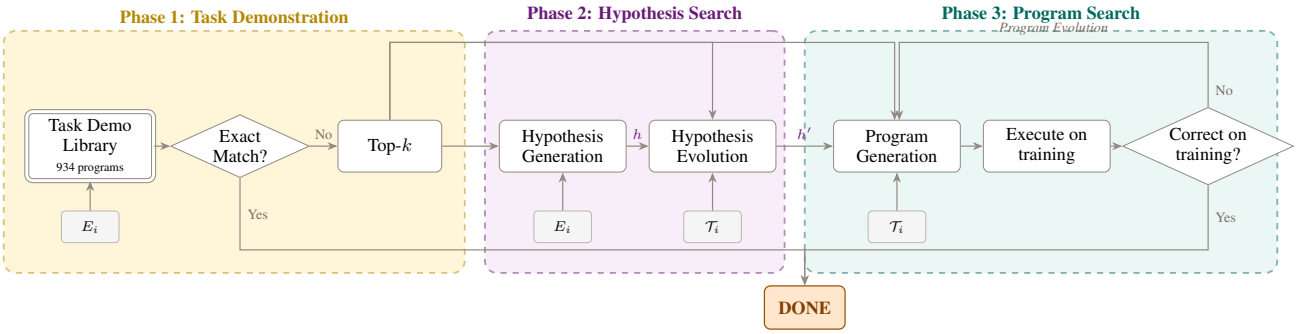

Figure 1. **Overview of the HINT pipeline. Phase 1 (Task Demonstration):** The task demonstration library (934 programs) is queried on the training examples E_i. An exact match terminates the pipeline immediately. Otherwise, the top-k programs and their corresponding task images are chosen by functional similarity (Eq. 1), addressing the *experience gap*. **Phase 2 (Hypothesis Search):** A VLM generates a natural-language hypothesis h conditioned on E_i and the relevant programs (Hypothesis Generation), then evolves it to h' by conditioning on the full task $\mathcal{T}_i$ — training examples and test input grid only, with no access to the test output (Hypothesis Evolution), addressing the *reasoning gap*. **Phase 3 (Program Search):** A VLM synthesizes a candidate program conditioned on h' and $\mathcal{T}_i$ (Program Generation). If the program fails on any training example, it is evolved by feeding back the execution error alongside h', anchoring repair to the intended transformation semantics rather than surface symptoms (Program Evolution).

synthesis, without grounding context, it requires excessive sampling as LLMs cannot reliably self-correct stochastic reasoning errors (Huang et al., 2024), making it both expensive and brittle. We call this the *reasoning gap*. We show that addressing these seemingly independent gaps alone is insufficient for consistent accuracy gains under cost constraints.

Under the Bayesian view of In-Context Learning (ICL) (Xie et al., 2021), task demonstrations concentrate the posterior over latent transformation rules, narrowing the space of plausible transformations to a tractable neighborhood. However, narrowing the search space does not lead to a better search algorithm. As Wang et al. (2024b) observe, natural language offers abstract but ambiguous representations while programs are detailed but verifiable. Prior experience adds up synergistically into exact accurate transformations when paired with a coarse-to-fine hierarchical reasoning and abstract hypothesis in natural language, followed by program synthesis (Merrill & Sabharwal, 2024; Press et al., 2023).

Motivated by these findings, we propose HINT 1, a framework for Hierarchical INference with Task-demonstrations that operationalizes prior experience and hierarchical reasoning for ARC problem solving. HINT comprises three components: (i) a curated library of approximately 1,000 ARC solutions, from which structurally similar examples act as past experience, (ii) a natural-language hypothesis generation and refinement phase that formulates solutions on an abstract level, and (iii) a formalization phase that translates the natural language hypothesis into an executable Python program. Together, these components provide a structured framework to address both gaps as prior experience grounds hypothesis formation, and hierarchical reasoning assembles relevant experience into executable solutions.

Despite relying entirely on off-the-shelf foundation models without fine-tuning, HINT achieves strong results across a range of cost levels on ARC-AGI-2. Specifically, HINT attains 39.2% accuracy at \$0.83 per task, 37.5% accuracy at \$0.63 per task, and 20.8% accuracy at \$0.28 per task under different configurations without finetuning and additional datasets. Additionally, we show that the combination of prior experience and hierarchical reasoning works synergistically and accounts for the bulk of the performance gains, with neither component explaining more than a fraction individually. These results provide evidence for the hypothesis that hierarchical reasoning and principled reuse of prior experience are critical for efficient abstract reasoning. Overall, our contributions are:

- We identify two methodological gaps in existing inductive approaches to ARC, an experience gap and a reasoning gap, and show empirically that they are mutually dependent and must be addressed together.
- We introduce HINT, a hierarchical inference framework that addresses both gaps through principled reuse of prior experience and hypothesis search, without any model fine-tuning or additional training datasets.
- We demonstrate the importance of natural language abstract reasoning within HINT. Replacing only the hypothesis-formation phases with a stronger model recovers nearly full performance at half the sampling budget under similar costs, suggesting inference-time design is a productive alternative to model scaling for compositional reasoning.

2. Related Works

2.1. Approaches to ARC

Approaches to solve ARC can be categorized as inductive or transductive approaches (Li et al., 2025). Inductive approaches try to infer the underlying transformation, like an executable program, from the training examples, which is then applied to the test input. In contrast, transductive approaches reason directly from the training examples to the test input to produce the output. We focus exclusively on inductive approaches in this work as we are inferring the underlying transformation.

Early inductive approaches relied on hand-designed, interpretable domain-specific languages combined with exhaustive or heuristic search with limited scalability (Vikram, 2020). While interpretable, these methods scale poorly to the diversity of ARC transformations. More recent methods use LLMs to generate candidate programs or abstract hypotheses that substantially expand the expressivity of the search space (Greenblatt, 2024; Pourcel et al., 2025; Berman, 2024; Pang, 2025). Among them, Hypothesis Search (Wang et al., 2024b) is most conceptually related to our work. It searches over a space of natural language hypotheses before committing to a program, demonstrating that explicit natural language abstraction can outperform direct prediction on ARC tasks. However, Hypothesis Search is extremely expensive, needs human-curated seed hypotheses to perform well, and treats each task in isolation.

2.2. Natural Language Abstraction for Program Synthesis

Recent works show that natural language abstraction substantially improves LLM program synthesis (Jiang et al., 2024) and that the natural language idea space is more tractable to explore than code space directly (Wang et al., 2024a). Along similar lines, Zelikman et al. (2023) show that hierarchical natural language decomposition into verified sub-functions outperforms end-to-end generation. These results establish natural language as the appropriate space for hypothesis search and building formal code over the constrained space for verification, motivating our two-level abstraction that underlies HINT.

However, natural language hypothesis formation is brittle without grounding. In settings like ARC, where transformation rules are arbitrary compositions of visual and spatial concepts, forming hypotheses from first principles means reasoning over an unbounded space. Reasoning chains in standard instruction-tuned models risk being unfaithful to the underlying task, producing plausible-sounding but incorrect hypotheses (Turpin et al., 2023; Lanham et al., 2023). Even in stronger models, early reasoning steps can disproportionately anchor the trajectory of subsequent reasoning (Bogdan et al., 2025). Without concrete evidence of how similar transformations have been implemented before, hypothesis formation over an unbounded space is likely to produce confidently wrong rules. This suggests that natural language abstraction needs to be anchored in relevant prior experience to be reliable, a gap that HINT addresses.

2.3. Experience Reuse and Task Demonstrations

The quality of relevant experience matters as much as its presence. From a Bayesian perspective on ICL, demonstrations shift the posterior over latent task structure when demonstrations share latent structure with the query. This motivates selective retrieval as functionally similar demonstrations are more likely to concentrate the posterior toward the correct transformation concept than arbitrarily chosen ones.

In the context of ARC literature, SOAR Pourcel et al. (2025) uses experience replay to iteratively fine-tune a model on previously solved tasks, thereby improving performance at the cost of inference-time efficiency due to repeated parameter updates. On the other hand, Pang (2025)'s architecture maintains maintains a library and retrieves related programs to guide generation, demonstrating that prior experience helps. However, both approaches use prior experience as an implicit signal either encoded into model parameters through fine-tuning, or provided as passive context without a structured mechanism to enforce its use during reasoning.

Together, these threads show that neither natural language abstraction nor retrieval alone is sufficient. HINT addresses both by treating prior experience as a structured inductive bias that actively shapes reasoning at multiple levels of abstraction.

3. Methodology

We propose HINT (Hierarchical INference with Task-demonstrations; Figure 1), a framework that enhances inductive reasoning by exploiting compositional domain regularities at different levels of abstractions. HINT maintains a library of previously attempted tasks and structures inference as a sequence of reasoning phases that enable principled reuse of prior experience across tasks.

3.1. Problem Formulation

ARC is a benchmark consisting of few-shot reasoning tasks formulated as small grid-based puzzles. Each task, $\mathcal{T}_i$ is defined as a tuple $\mathcal{T}_i = (E_i, x_i^{\text{test}})$, where $E_i = \{(x_{i,j}^{\text{train}}, y_{i,j}^{\text{train}})\}_{j=1}^{N}$ is a set of N input-output training samples and x_i^{test} is a test input grid.

We treat this as a programming by example (PBE) problem where the goal is to find a Python program that defines the

function $f : \mathbb{G} \to \mathbb{G}$ (where $\mathbb{G}$ is the space of integer grids) s.t. $y_{i,j}^{\text{train}} = f(x_{i,j}^{\text{train}})$ for all $(x_{i,j}^{\text{train}}, y_{i,j}^{\text{train}}) \in E_i$, and which generalizes to produce the correct test output $\hat{y} = f(x_i^{\text{test}})$.

Each ARC task has a unique, exact transformation function f composed of fundamental concepts from intuitive physics, basic arithmetic, pattern completion, or spatial reasoning. While individual tasks may appear simple for humans, their compositional structure makes ARC a challenging benchmark for abstraction and reasoning.2

3.2. Phase 1: Task Demonstration

The experience gap arises because models that treat each task in isolation must search over an unbounded concept space from scratch, discarding solutions to structurally related problems they have already encountered. Task Demonstration directly addresses this by maintaining a dynamic library $\mathcal{D}$ containing previously solved tasks and their verified solution programs, where d_k represents a solution program in the library. The initial library consists of 934 programs compiled from two sources:

- **Exact Solutions** Hand-written DSL-based solutions for the ARC-1 training dataset, where each solution corresponds to an exact task
- **Approximate Solutions** Attempted solutions for the ARC-2 training dataset

Functional similarity retrieval A natural approach would be to choose tasks that visually resemble the current task. However, we do not rely on visual similarity as it is a poor proxy for structural relatedness and can be misleading. Instead, we use functional similarity by executing programs from the library on the current task's examples. We measure similarity functionally by executing the library $\mathcal{D}$ directly on the current task's training examples and measuring how well their outputs match.

For a new task $\mathcal{T}_i$ with examples E_i, we execute every program $d_k \in \mathcal{D}$ on the input grids $x_{i,j}^{\text{train}} \in E_i$. We measure the similarity of a library program d_k with our current task T_i using a similarity score based on pixel-wise accuracy:

$$S(d_k, E_i) = \frac{1}{|E_i|} \sum_{(x,y) \in E_i} \text{match}(d_k(x), y) \qquad (1)$$

where match$(\cdot, \cdot)$ computes the grid similarity (handling dimension mismatches via padding). Programs that yield $S = 1.0$ are immediate exact solutions. If no exact match is found, we rank the top-K tasks from each source with the highest scores, forming $\mathcal{D}_{sim} = \{d_1, \ldots, d_K\}$.

In cases where we have exact solutions for ARC-1 tasks, we pass images of those tasks as a visual reference. This grounds the downstream phases in concrete examples of how structurally similar transformations present visually. For the approximate solutions, we pass the source code as the implementation logic may share compositional concepts with the current task. By ranking and filtering the library based on functional similarity, HINT ensures that relevant experience narrows the posterior over plausible transformations, directly closing the experience gap.

3.3. Phase 2: Hypothesis Search

Using the functionally grounded set of demonstrations, Phase 2 addresses the reasoning gap. With it, the model reasons over a concept space that has already been partially charted by similar solved tasks. HINT induces and refines an abstract natural-language hypothesis about the transformation to uncover key features in natural language that are difficult to directly verify through program synthesis.

Hypothesis Generation (h): We prompt a Vision-Language Model (VLM) with a dynamic prompt consisting of (i) the training examples E_i rendered as both base64 images and ASCII grids, (ii) the relevant tasks, $\mathcal{D}_{sim}$ with their similarity scores, the source code of the approximately similar programs and, for exact solutions, their corresponding task images and (iii) instructions asking the model to analyze the training examples sequentially, iteratively refine its hypothesis, and provide a final description of the transformation rule in natural language. By providing only training samples and similar tasks, we nudge the VLM to discover underlying patterns consistent with both the current task and with the demonstrations.

Hypothesis Evolution (h') The initial hypothesis h is strictly derived only from training data and may overfit to the specific configurations seen in E_i. The VLM evolves $h \to h'$ by conditioning on the test input x_i^{test} to ensure the logic generalizes to test input as well to produce a more robust description. This mimics how a human solver might revisit their initial interpretation after seeing the actual test case, catching overfits to the training examples before they propagate into code.

3.4. Phase 3: Program Search

Phase 3 translates the evolved hypothesis h' into an executable Python function f, closing the reasoning gap. By conditioning program synthesis on h' and $\mathcal{D}_{sim}$ instead of direct program synthesis, HINT encourages structured reasoning accumulated across Phases 1 and 2 to shape code generation, thus making the search over program space significantly more constrained and directed.

Program Synthesis We construct a prompt containing the examples E_i, the test input x_i^{test}, top-k similar tasks $\mathcal{D}_{sim}$, and the evolved hypothesis h'. The VLM then generates a

candidate function

$$f \sim P(\cdot \mid E_i, x_i^{\text{test}}, \mathcal{D}_{sim}, h').$$

Program Evolution If the generated function f fails on the training examples (i.e., $f(x) \neq y$ for some $(x, y) \in E_i$), we perform a refinement step. We feed the execution error (the difference between predicted output and ground truth) back to the model, prompting it to debug the code while anchored to the intended transformation semantics. The idea behind this is that it prevents the model from making surface-level fixes without increasing the costs substantially. This yields a refined function f^+:

$$f^+ \sim P(\cdot \mid f, \text{Error}(f, E_i), h')$$

The exact prompts for all the stages of HINT are available in the Appendix B

3.5. Experimental Setup

Dataset. All results are reported on the ARC-AGI-2 public evaluation dataset (120 tasks) unless mentioned otherwise.

Baselines. Given the rapid pace of model releases, baseline comparisons are restricted to systems available as of February 15, 2026, corresponding to the completion of our experiments. As shown in Table 3, we focus on the sub-$1 per task regime as the relevant comparison point for efficient inference, where the strongest leaderboard result is 55.8% at $0.68.

Evaluation Protocol. Following the official ARC-AGI-2 evaluation protocol, each task permits two submission attempts(ARC Prize Foundation, 2025). To exploit this, we run k parallel samples of the full HINT pipeline per task and select the two highest-scoring candidate programs based on pixel-wise accuracy on the training examples (Eq. 1). Unless otherwise stated, we use k=4 as the default configuration. All reported accuracies are the median over 3 independent runs of the full pipeline to account for stochasticity in model sampling.

4. Experimental Results

Our experiments address two questions: (1) are hierarchical reasoning and retrieval individually sufficient, or is their combination necessary? (2) does HINT accuracy–cost scale as backbone model quality increases? Prior experience and structured reasoning seem like complementary approaches to ARC, but our experiments show that learning from prior experience and reasoning in natural language don't work well in isolation. We find that a model with access to similar solved tasks but no reasoning scaffold cannot reliably assemble that experience into a correct solution, and that a model with a careful reasoning process but no grounding tends to generate confident, coherent, and often incorrect hypotheses. We observe consistent gains when prior experience shapes reasoning at all levels, including natural language abstraction and program synthesis. We further analyze which tasks HINT solves and find that gains are concentrated in harder tasks, suggesting that hierarchical grounding provides the most benefit precisely where single-pass uninformed inference struggles most as can seen in this chart 3.

Model	Setting	Acc. (%)	Cost ($/task)
GPT-5.2	Direct program synthesis	20.0	0.47
Claude Sonnet 4.5	Direct program synthesis	13.3	0.75
Grok 4.1-Fast (High)	Direct program synthesis	5.0	0.08
Grok 4.1-Fast (Med)	Direct program synthesis	3.3	0.06
Grok 4.1-Fast (k=4)	2-phase hyp. search	11.7	0.23
Grok 4.1-Fast (k=8, more parallel sampling)	2-phase hyp. search	15.0	0.52

Table 1. Direct program synthesis and hypothesis search on ARC-AGI-2. Hypothesis search improves over direct program synthesis under matched compute, motivating the need to address the reasoning gap. All numbers are from our own experimental runs under identical evaluation conditions.

4.1. Models must leverage hypothesis search to address the reasoning gap

Can state-of-the-art language models solve ARC tasks inductively in a single attempt? Table 1 reports results from our own experimental runs under identical evaluation conditions, unless stated otherwise. Even strong models struggle: Grok 4.1-Fast achieves 5% in a single-shot setting. Under a moderate compute budget of under $1 per task, most large models achieve modest success rates on ARC, e.g., GPT-5.2 (20%), while smaller models perform even worse, with Grok 4.1-Fast achieving 3%-6% across different reasoning capabilities. Direct program synthesis in a low-data regime is still too challenging for current language models.

This lack of performance can be addressed through hierarchical hypothesis search, in which language is used as an abstract representation, grounding program synthesis, and leading to higher accuracy. We compare two settings with Grok4.1-Fast to sample candidate programs: 1) single-shot sampling and 2) vanilla 2-phase hypothesis search. Table 1 shows that a hierarchical hypothesis formation improves performance. Notably, this allows much cheaper models to outperform much larger ones on a similar budget. This achieves 15% accuracy on a budget of $0.52 per task and beats Claude-4.5-Sonnet. Additionally, we also compare k=8 and k=4. It shows that additional sampling doesn't compensate for the absence of retrieved experience, thus motivating the need for HINT.

However, this process requires excessive sampling because the sequential nature of hypothesis search introduces stochastic errors in reasoning traces (Lanham et al., 2023). Without prior experience to constrain the hypothesis space, the model generates plausible but unfaithful reasoning

chains that fail to converge on the correct rule (Turpin et al., 2023). This motivates our key question: can we use shared concepts from relevant past experiences for efficient hierarchical hypothesis search? Our experiments show that using prior experience with hierarchical hypothesis search has synergistic effects and leads to higher accuracy at the same cost.

4.2. HINT: Hierarchical INference with Task-demonstrations

Grounding hypothesis search in retrieved prior experience improves accuracy without a sharp increase in costs. Using Grok 4.1-Fast as the low-cost model in the pipeline, HINT achieves 20.8% accuracy at $0.28 per task, compared to 11.7% for ungrounded hypothesis search at a similar cost. It also matches the accuracy achieved by GPT 5.2 at 40% lower costs.

Learning from relevant prior experience. A key insight for ARC is the compositional nature of the problem. All transformations are compositions of common concepts like rotation and mirroring. Treating each ARC task as an isolated problem and not reusing previously discovered concepts makes the search inefficient. We find that replacing functionally similar retrieved programs with randomly chosen ones yields 16.7% accuracy, which is identical to having no retrieval at all. The mere presence of programs without context relevance provides no benefit. Random programs add noise to the hypothesis formation context without narrowing the posterior over transformation concepts, consistent with the Bayesian interpretation of in-context learning (Xie et al., 2021).

Additionally, we observe that providing similar programs only during the synthesis phase, while leaving hypothesis formation ungrounded, recovers some performance (18.3%) but falls short of full HINT (20.8%). The hypothesis formed without grounding remains partially unconstrained, but without shaping the natural language abstraction upstream, the program synthesis phase receives a weaker signal.

Hypothesis evolution allows for the model to self-correct. As we have already seen, errors in LLM reasoning compound across phases even for very simple tasks during hypothesis search. Splitting into generation followed by evolution allows the model to revise its understanding incrementally against training and test samples from a task. We also observe this when merging the generation and evolution phases into a single hypothesis step, which reduces accuracy to 15.8%, below even the ungrounded hypothesis search baseline.

Program Refinement adds consistent value. It was often observed during manual inspection that in some cases, program synthesis failed due to execution errors and corner cases, while the natural language reasoning captured the underlying pattern accurately. In such cases, doing one iteration of program evolution increased the accuracy from 19.2% to 20.8% with a mere increase of $0.03 per task.

We also observed that program refinement failed to improve programs, including those that already pass training examples. Instead, it led to a drop in accuracy to 16.7% from 20.8% while slightly increasing costs, introducing unnecessary edits to programs that generalize correctly. Repair is most effective when reserved for programs with execution failures, where the error signal is informative.

Together, these results show that HINT's gains require all components to work in tandem. Each component addresses a distinct failure mode: semantic retrieval constrains the hypothesis space, grounded hypothesis formation propagates that constraint through reasoning, hypothesis evolution allows for self-correction, and selective program refinement recovers from execution errors without overfitting. Removing any one of these consistently degrades performance, confirming that the components are complementary and collectively necessary rather than individually sufficient.

4.3. Reasoning quality disproportionately drives HINT's scaling gains

Our scaling experiments show that the natural language reasoning phases are the primary bottleneck. Using a hybrid approach where a stronger model performs natural language reasoning yields disproportionate gains, while synthesis can remain cheap without sacrificing much accuracy.

In a hybrid configuration, we achieve the same accuracy (20.8%) at comparable costs, with half the sampling when using Gemini 3 Flash for hypothesis formation and refinement while retaining Grok 4.1-Fast for program synthesis. A stronger reasoning model compensates for reduced sampling, suggesting that hypothesis quality matters more than hypothesis quantity and the errors compound in the reasoning phase.

Increasing the number of samples to k=4 in the above-explained hybrid setting pushes accuracy to 37.5% at $0.63 per task, nearly doubling performance while roughly doubling the cost of the Grok-only baseline. On the other hand, using Gemini 3 Flash throughout the pipeline reaches 39.2% at $0.83 per task, a marginal improvement over the hybrid at a much higher cost. This empirically shows that a good natural language hypothesis can guide program synthesis even with a weak model.

5. Discussion and Conclusion

Our work shows that efficient abstract reasoning requires better inference-time structure without finetuning and larger models. Efficient reasoning under novel conditions without

LLM	Retrieval	Split	Repair	Acc. (%)	Cost ($/task)
Ungrounded hypothesis search (Reference starting point)					
Grok 4.1-Fast	None	2ab	×	11.7	0.23
Effect of hypothesis formation strategy					
Grok 4.1-Fast	**Similar**	**2a+2b**	✓	**20.8**	**0.28**
Grok 4.1-Fast	Similar	2ab	✓	15.8	0.26
Grok 4.1-Fast	Similar	2a+2b	×	19.2	0.25
Effect of retrieval quality					
Grok 4.1-Fast	**Similar**	**2a+2b**	✓	**20.8**	**0.28**
Grok 4.1-Fast	Synthesis-only	2a+2b	✓	18.3	0.32
Grok 4.1-Fast	Random	2a+2b	✓	16.7	0.29
Grok 4.1-Fast	None	2a+2b	✓	16.7	0.28
Grok 4.1-Fast	Similar	2a+2b	Over-repair	16.7	0.30
Effect of backbone model (full HINT pipeline)					
Grok 4.1-Fast	Similar	2a+2b	✓	20.8	0.28
Flash + Grok 4.1-Fast† (k=2)	Similar	2a+2b	✓	20.8	0.32
Flash + Grok 4.1-Fast† (k=4)	Similar	2a+2b	✓	37.5	0.63
Gemini 3 Flash (full)†	Similar	2a+2b	✓	**39.2**	0.83

Table 2. Ablation results for HINT on ARC-AGI-2 (public evaluation set, 120 tasks). All rows use Grok 4.1-Fast as backbone unless otherwise indicated; k=4 throughout unless noted. *Split*: whether hypothesis formation is split into generation (2a) and evolution (2b) or merged into a single stage (2ab). *Retrieval*: whether retrieved tasks are functionally similar, randomly chosen, absent, or provided only at the synthesis stage. *Repair*: whether program evolution (execution-guided refinement) is applied to programs that fail on training examples. Results marked with † use a stronger backbone model.

task-specific supervision remains one of the central open problems in AI. ARC-AGI-2 tests more advanced AI capabilities, such as symbolic interpretation and compositional reasoning. These tasks are easy for humans because we draw from our rich prior experience and reason hierarchically, whereas most AI systems treat each problem in isolation and search over an unconstrained hypothesis space. HINT shows that grounding hypothesis formation in relevant prior experience and decomposing reasoning into a coarse-to-fine hierarchy are synergistic. While neither component alone produces consistent gains, together they enable a cheap model to match the accuracy of much larger models at a fraction of the cost.

Our experiments reveal two complementary findings. In line with the Bayesian view on ICL, demonstrations improve inference only when they share latent structure with the query. We also observe that natural language reasoning phases are the primary bottleneck in multi-phase inference. Replacing only the reasoning backbone with a stronger model recovers performance with half the sampling budget, confirming that hypothesis quality matters more than hypothesis quantity. Together, these findings suggest that inference-time design is a productive alternative to model scaling for compositional reasoning tasks.

HINT also highlights an underexplored dimension of experience reuse. Existing approaches either encode prior experience into model parameters through extensive fine-tuning or lack a structured mechanism to enforce grounded reasoning to actively use prior experience. HINT instead routes prior experience through an explicit natural language hypothesis synthesis phase, using experience as a structured inductive bias at multiple levels of abstraction. This allows HINT to benefit from retrieval without any parameter update.

A likely bottleneck is the static library at evaluation time, which limits the compounding benefit of experience over a long task sequence. A natural extension is an online version of HINT, in which successful solutions are dynamically added to the library. However, online library updates require sequential execution, making evaluation prohibitively slow at the current scale. While our primary results use Grok 4.1-Fast as the backbone, and our hybrid experiments show that hypothesis quality improves with stronger reasoning models, a systematic evaluation across a broader range of models remains future work. More broadly, HINT's design principle of grounding structured reasoning in relevant prior experience is not specific to ARC. The same architecture can be extended to any domain where tasks share compositional structure and verified solutions can be stored, such as mathematical problem solving or structured code generation.

References

ARC Prize Foundation. Arc prize. https://arcprize.org, 2025. Accessed: 2026-02-15.

Berman, J. arc_agi. https://github.com/jerber/arc_agi, 2024. Accessed: November 4, 2025.

Berman, J. arc-lang-public. https://github.com/jerber/arc-lang-public, 2025. Accessed: November 4, 2025.

Bogdan, P. C., Macar, U., Nanda, N., and Conmy, A. Thought anchors: Which LLM reasoning steps matter?, 2025. URL https://arxiv.org/abs/2506.19143.

Chollet, F. On the measure of intelligence, 2019. URL https://arxiv.org/abs/1911.01547.

Greenblatt, R. Getting 50 URL https://blog.redwoodresearch.org/p/getting-50-sota-on-arc-agi-with-gpt. Blog post.

Huang, J., Chen, X., Mishra, S., Zheng, H. S., Yu, A. W., Song, X., and Zhou, D. Large language models cannot self-correct reasoning yet. In *International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=IkmD3fKBPQ.

Jiang, X., Dong, Y., Wang, L., Fang, Z., Shang, Q., Li, G., Jin, Z., and Jiao, W. Self-planning code generation with large language models. *ACM Transactions on Software Engineering and Methodology*, 33(7):1–30, 2024.

Lanham, T., Chen, A., Radhakrishnan, A., Steiner, B., Denison, C., Hernandez, D., Li, D., Durmus, E., Hubinger, E., Kernion, J., et al. Measuring faithfulness in chain-of-thought reasoning. *arXiv preprint arXiv:2307.13702*, 2023.

Li, W.-D., Hu, K., Larsen, C., Wu, Y., Alford, S., Woo, C., Dunn, S. M., Tang, H., Zheng, W.-L., Pu, Y., and Ellis, K. Combining induction and transduction for abstract reasoning. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=UmdotAAVDe.

Merrill, W. and Sabharwal, A. The expressive power of transformers with chain of thought. In *International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=NjNGlPh8Wh.

Pang, E. arc_agi. https://github.com/epang080516/arc_agi, 2025. Accessed: November 4, 2025.

Pourcel, J., Colas, C., and Oudeyer, P.-Y. Self-improving language models for evolutionary program synthesis: A case study on ARC-AGI. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, Vancouver, Canada, 2025. PMLR. URL https://openreview.net/forum?id=z4IG090qt2.

Press, O., Zhang, M., Min, S., Schmidt, L., Smith, N., and Lewis, M. Measuring and narrowing the compositionality gap in language models. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 5687–5711, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.findings-emnlp.378. URL https://aclanthology.org/2023.findings-emnlp.378.

Spelke, E. S. and Kinzler, K. D. Core knowledge. *Developmental Science*, 10(1):89–96, January 2007. ISSN 1363-755X, 1467-7687. doi: 10.1111/j.1467-7687.2007.00569.x. URL https://onlinelibrary.wiley.com/doi/10.1111/j.1467-7687.2007.00569.x.

Turpin, M., Michael, J., Perez, E., and Bowman, S. Language models don't always say what they think: Unfaithful explanations in chain-of-thought prompting. *Advances in Neural Information Processing Systems*, 36: 74952–74965, 2023.

Vikram, V. ARC-icecuber: Code for 1st place solution to kaggle's abstraction and reasoning challenge. GitHub repository, 2020. URL https://github.com/victorvikram/ARC-icecuber. MIT License; forked from top-quarks/ARC-solution.

Wang, E., Cassano, F., Wu, C., Bai, Y., Song, W., Nath, V., Han, Z., Hendryx, S., Yue, S., and Zhang, H. Planning in natural language improves llm search for code generation. *arXiv preprint arXiv:2409.03733*, 2024a.

Wang, R., Zelikman, E., Poesia, G., Pu, Y., Haber, N., and Goodman, N. Hypothesis search: Inductive reasoning with language models. In *The Twelfth International Conference on Learning Representations*, 2024b. URL https://openreview.net/forum?id=G7UtIGQmjm.

Xie, S. M., Raghunathan, A., Liang, P., and Ma, T. An explanation of in-context learning as implicit bayesian inference. *arXiv preprint arXiv:2111.02080*, 2021.

Zelikman, E., Huang, Q., Poesia, G., Goodman, N., and Haber, N. Parsel: Algorithmic reasoning with language models by composing decompositions. *Advances in Neural Information Processing Systems*, 36:31466–31523, 2023.

A. Appendix

Model	Solved	Score (%)	Cost ($)
Gemini 3.1 Pro Preview	67.0	55.8	0.7
Gemini 3 Flash (High)	34.0	28.3	0.2
Gemini 3 Pro	28.0	23.3	0.8
Gemini 3 Flash (Med)	16.0	13.3	0.1
Gemini 3 Flash (Min)	2.0	1.7	0.0
Gemini 3 Flash (Low)	1.0	0.8	0.0
GPT-5.2 (XHigh)	59.0	49.2	1.8
GPT-5.2 Pro (High)	58.0	48.3	16.5
GPT-5.2 (High)	41.0	34.2	1.5
GPT-5.2 Pro (Med)	40.0	33.3	9.5
GPT-5.2 (Med)	26.0	21.7	0.8
GPT-5.1 (High)	18.0	15.0	1.2
GPT-5.2 (Low)	8.0	6.7	0.3
GPT-5.1 (Med)	8.0	6.7	0.5
GPT-5.1 (Low)	1.0	0.8	0.1
Claude Opus 4.5 (64K)	43.0	35.8	2.5
Claude Opus 4.5 (32K)	32.0	26.7	1.4
Claude Opus 4.5 (16K)	24.0	20.0	0.8
Claude Opus 4.5 (8K)	10.0	8.3	0.5
Claude Opus 4.5 (None)	8.0	6.7	0.2
Claude Opus 4.5 (1K)	6.0	5.0	0.2
Grok 4.1 (Fast)	6.0	5.0	0.1
Grok 4 (Fast)	0.0	0.0	0.1

Table 3. ARC-AGI-2 benchmark results across frontier models on the evaluation set. Models are grouped by family and sorted by score descending within each group. Table generated on 15.02.2026

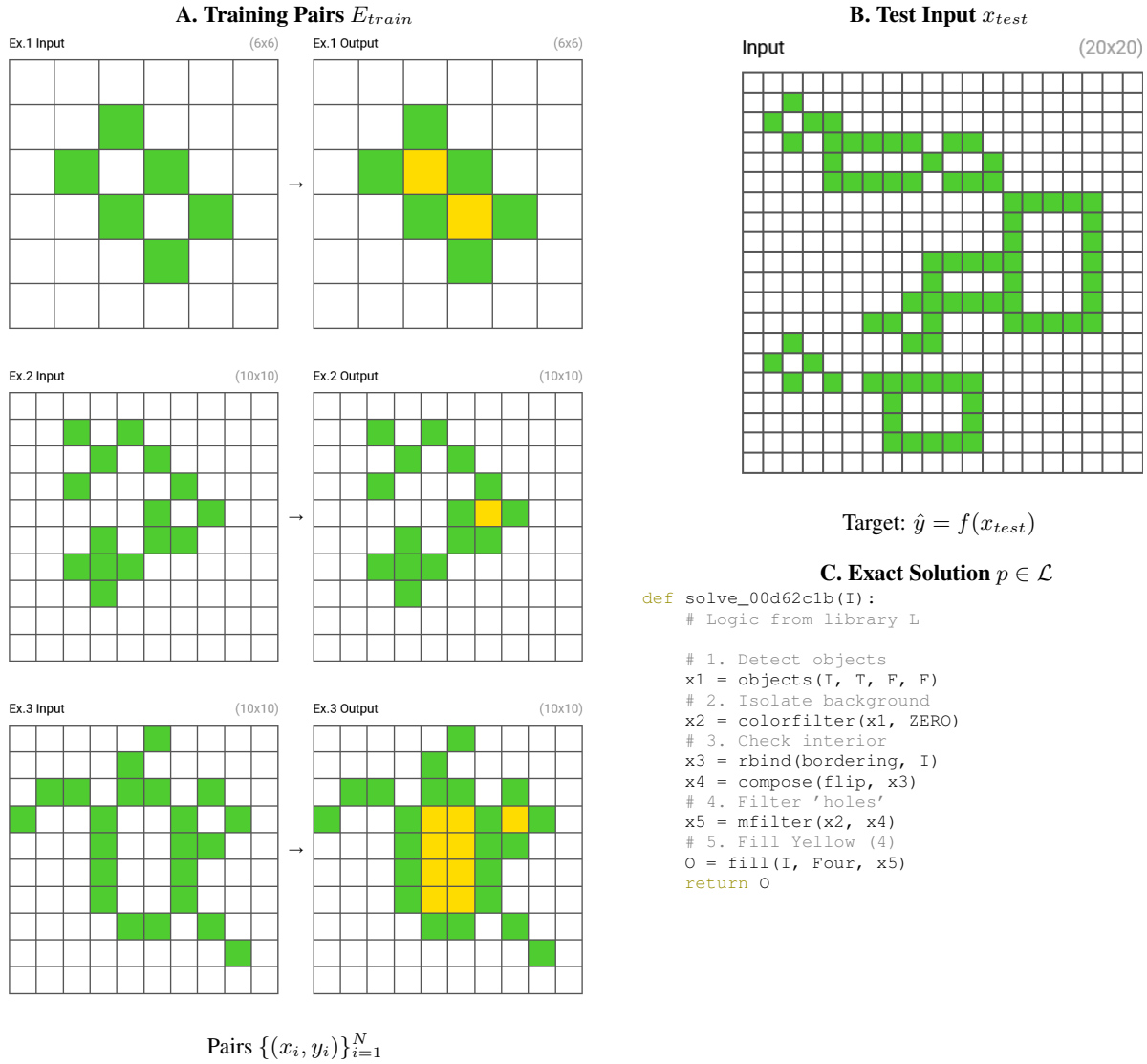

Figure 2. **ARC Task Structure and Library Solution.** (A) Training pairs E_{train} d demonstrating the input-output transformation. (B) Test input x_{test} for which the output must be predicted. (C) The corresponding exact solution p found in the HINT program library $\mathcal{L}$: the program detects enclosed regions in the input and fills them with color 4 (yellow), matching the transformation shown in (A). HINT retrieves such programs in Task Demonstration Stage to utilize prior experience for hypothesis formation.

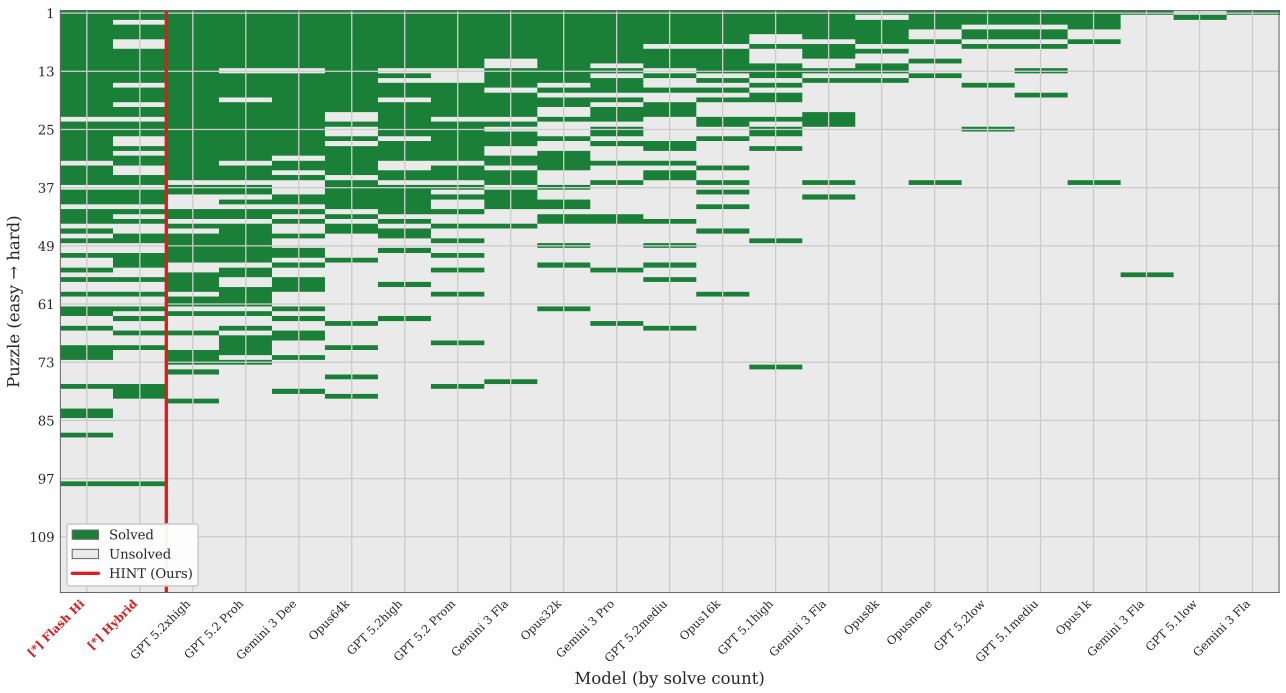

Figure 3. **Per-task solve comparison across frontier models and HINT.**Each row is a puzzle (sorted easy to hard top to bottom); each column is a model or configuration sorted by total solve count. Green indicates solved; white indicates unsolved. The red line marks HINT. HINT solves several puzzles in the harder tiers (rows 60–85) that no other model in this comparison solves, suggesting that hierarchical grounding provides the most benefit precisely where single-pass inference fails.

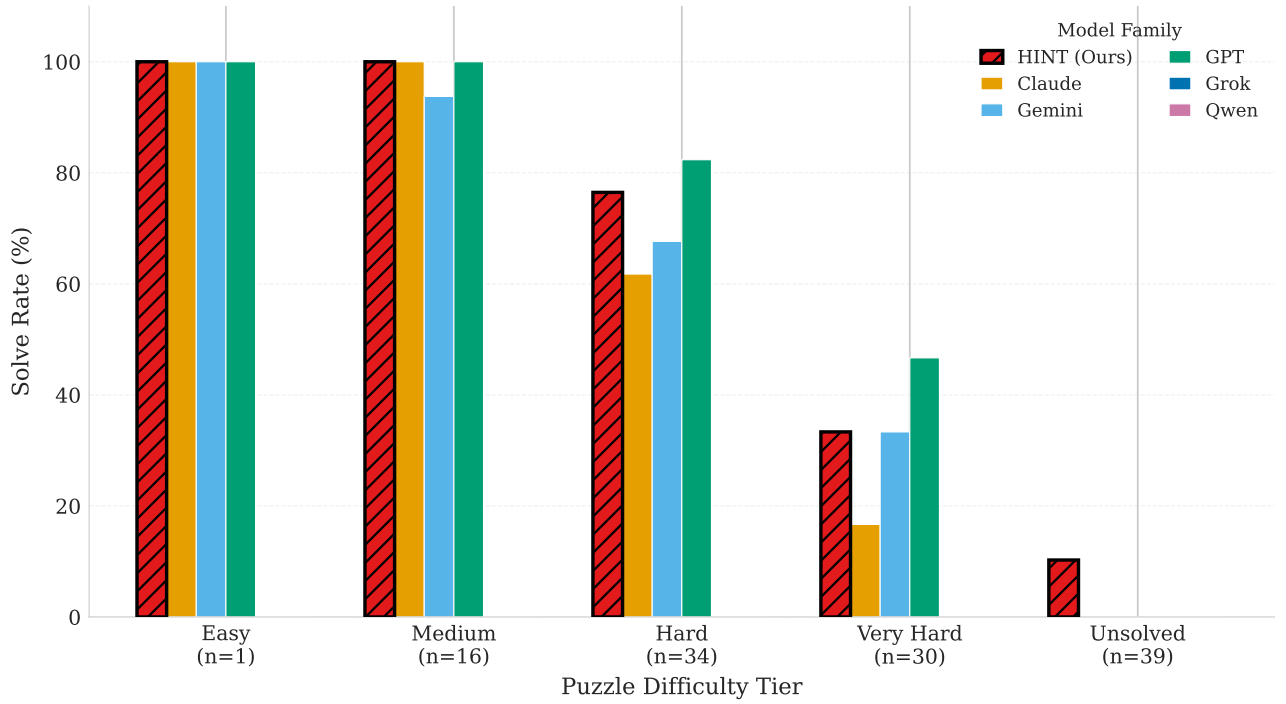

Figure 4. **Solve rate by puzzle difficulty tier**Puzzles are grouped into five tiers based on how many frontier models solve them (Easy through Unsolved). HINT (hatched red) matches or exceeds top models on Easy and Medium tiers, and maintains competitive solve rates on Hard and Very Hard tiers where direct-prompting approaches degrade most sharply.

B. Prompt Library

This appendix provides the complete prompts used in each phase of HINT. Each prompt corresponds to a stage in the pipeline described in Section 3: Hypothesis Generation (B.1) and Hypothesis Evolution (B.2) implement Phase 2, while Program Generation (B.3) and Program Evolution (B.4) implement Phase 3. The placeholders in curly braces indicate dynamic content filled at runtime.

B.1. Prompt 2a. Hypothesis Generation

The hypothesis generation prompt conditions the VLM on training examples and retrieved demonstrations to induce an initial natural-language hypothesis.

Prompt used in experiments

```
You are an expert at analyzing ARC
    puzzles and discovering
    transformation patterns.

Remember: Your first hypothesis is
    sticky and excessively convincing
    to you.
Combat this by evolving your hypothesis
    as you see each training example.

## Training Examples

{training examples}

## Similar Programs from Library

{similar programs}

## Analysis Protocol

You will analyze the examples
    systematically, allowing your
    hypothesis to evolve naturally as
    you see more data - like a human
    solving a puzzle.

**Core principle:** Look for
    DIFFERENCES within each example
    (inputoutput changes) and
    SIMILARITIES across all examples
    (the consistent pattern).



Work through examples sequentially,
    reasoning in simple English about
    what you observe.

**Step 1: First Example Analysis**
Given: Input 1  Output 1

<observation_1>
Describe what you see:
- What's the size/shape change?
- What colors changed?
- What geometric transformations
    occurred?
- What patterns do you notice?
</observation_1>

<hypothesis_1>
State your initial guess about the
    transformation rule in natural
    language.
Example: "The grid appears to be
    flipped horizontally"
</hypothesis_1>

**Step 2: Second Example Validation**
Given: Input 2  Output 2

<observation_2>
- Does your hypothesis from Example 1
    still hold?
- What's similar to Example 1?
- What's different from Example 1?
</observation_2>

<hypothesis_2>
Refine your hypothesis:
- If it still works: Confirm and
    strengthen
- If it breaks: Revise with a more
    general pattern
Example: "Actually, it's mirrored
    horizontally, THEN the original is
    stacked on top"
</hypothesis_2>

**Step 3: Third Example Confirmation**
Given: Input 3  Output 3

<observation_3>
- Does hypothesis_2 work here?
- Any new edge cases or variations?
</observation_3>

<hypothesis_3>
Final refined hypothesis in natural
    language.
</hypothesis_3>

**Step N: Additional Examples**
Continue for all remaining examples...

**Final Step: Pattern Synthesis**
<pattern_summary>
In plain English, the transformation
    rule is:
- [Short description of relevant
    observations]
- [First operation in natural language]
- [Second operation in natural language]
- [Any conditions or special cases]

Edge cases to consider:
```

```
- [Any variations you noticed]

Why this works:
- [Brief explanation about WHY this
    pattern makes sense]

Confidence level: [0-100%]
</pattern_summary>



Begin your analysis:
```

B.2. Prompt 2b. Hypothesis Evolution

The hypothesis evolution prompt refines the initial hypothesis by conditioning on the test input to catch overfits before program synthesis.

Prompt used in experiments

```
You are an expert at analyzing ARC
    puzzles and discovering
    transformation patterns.

You are given an initial hypothesis
    about the puzzle. If the hypothesis
    doesn't extend to the test input
    while explaining the training
    examples, refine it to create a
    more accurate pattern description.
Remember: Your first hypothesis is
    sticky and excessively convincing
    to you. The final transformation is
    a simple transformation that
    applies to all samples, both
    training and test.
Combat this by evolving your hypothesis

## Initial Hypothesis

{hypothesis}



## All Examples (Test + Train)

Your task: Test if this hypothesis
    works for test examples and train
    input output pairs below.
If it doesn't fit perfectly, identify
    what needs to be refined.

### Test Examples (inputs only)

{test examples}

### Training Examples

{training examples}

## Validation Protocol

Check if the initial hypothesis fits
    ALL examples (training + test
    inputs).
Your goal: Confirm the hypothesis or
    refine it as needed.

**For each test input:**
<test_check_N>
Does the hypothesis make sense for Test
    Input N?
- Consider: size, colors, patterns,
    edge cases
- Any concerns or refinements needed?
</test_check_N>

Does it explain all the test and
    training examples perfectly?

**Final Validation:**
<validated_pattern>
After checking all examples:

**Status:** [CONFIRMED / NEEDS
    REFINEMENT]

**Final Pattern Description:**
[If confirmed: restate the pattern
    cleanly]
[If refined: provide the improved
    pattern with explanations of what
    changed]

**Confidence:** [0-100%]

**Reasoning:** [Why this pattern works
    for all examples]



Begin validation:
```

B.3. Prompt 3a. Program Generation

The program generation prompt translates the evolved hypothesis into an executable DSL function, conditioned on retrieved similar programs.

Prompt used in experiments

```
You are an expert at generating code
    using the given DSL primitives to
    solve ARC puzzles. You are provided
    with a natural language description
    of the pattern to implement, as
    well as training and test examples
    and some similar programs you might
    find useful as reference. Generate
    a Python function `def solve(I):`
    that implements the described
```

```
    transformation using ONLY the
    provided DSL primitives. Ensure
    your code is syntactically correct
    and follows best practices

# DSL Code Generator\nGenerate a Python
    `solve(I)` function using ONLY the
    DSL primitives below.

## Training Examples

{training examples}

## Test Examples (inputs only)

{test examples}

## Natural Language Pattern Description

{pattern}

## Similar Programs for Reference

The following programs solved similar
    tasks and may provide useful
    patterns or approaches:

{similar programs}

## DSL Primitives

**Type Definitions:**
Grid: Tuple[Tuple[int, ...], ...] -
    Immutable 2D array (tuple of tuples)
Object: FrozenSet[(int, (int, int))] -
    Set of (color, location) pairs
Patch: FrozenSet[(int, int)] or Object
    - Set of indices or colored object
Indices: FrozenSet[(int, int)] - Set of
    (row, col) positions
Objects: FrozenSet[Object] - Set of
    objects
Container: Tuple or FrozenSet - Generic
    container type
IntegerTuple: (int, int) - Tuple of
    integers (usually coordinates or
    dimensions)
Callable: Function type

**Functional Programming:**
```python
compose(f, g) -> Callable
         # f(g(x)) - function
    composition
chain(f, g, h, ...) -> Callable
         # f(g(h(...(x)))) - chain
    multiple functions
combine_two_function_results(combiner,
    f, g) -> Callable # combiner(f(x),
    g(x)) - apply two functions and
    combine
transform(f, container) -> Container
         # apply f to each element
transform_and_flatten(f, container) ->
    FrozenSet # apply f over container
    followed by merge results
fix_first_argument(f, arg) -> Callable
         # f(arg, x) - left partial
    application
fix_last_argument(f, arg) -> Callable
         # f(x, arg) - right partial
    application
identity(x) -> Any
         # returns x unchanged
equals(f, container) -> Callable
         # find element in container
    where f matches
extract_first_matching(container, f) ->
    Any  # first element satisfying
    predicate f
```

Grid Transforms:
```python
horizontal_mirror(grid) -> Grid/Patch
         # flip left-right
vertical_mirror(grid) -> Grid/Patch
         # flip up-down
diagonal_mirror(grid) -> Grid/Patch
         # diagonal \\ mirror
counterdiagonal_mirror(grid) ->
    Grid/Patch   # diagonal / mirror
rot90(grid) -> Grid
         # rotate 90° clockwise
rot180(grid) -> Grid
         # rotate 180°
rot270(grid) -> Grid
         # rotate 270° clockwise
vertical_concat(a, b) -> Grid
         # stack vertically [a; b]
horizontal_concat(a, b) -> Grid
         # stack horizontally [a, b]
crop(grid, (i,j), (h,w)) -> Grid
         # extract subgrid starting at
    (i,j) with dimensions (h,w)
upscale(grid, n) -> Grid/Object
         # enlarge by factor n
downscale(grid, n) -> Grid
         # shrink by factor n
horizontal_split(grid, n) -> Tuple
         # split horizontally into n
    parts
vertical_split(grid, n) -> Tuple
         # split vertically into n parts
top_half/bottom_half -> Grid
         # get top/bottom half
left_half/right_half -> Grid
         # get left/right half
trim_border(grid) -> Grid
         # remove border
```

Objects:
```python
as_objects(grid, univalued, diagonal,
    without_bg) -> FrozenSet[Object]
```
```

```
# Find connected components (separate
    shapes/regions)
# univalued: T = single-color objects,
    F = multicolor allowed
# diagonal: T = diagonal connects
    (8-connected), F = only orthogonal
    (4-connected)
# without_bg: T = ignore background
    (most common color)
# Returns: frozenset of objects

color_filter(objects, color) ->
    FrozenSet[Object] # Keep only
    objects of specified color
size_filter(objects, n) -> FrozenSet
              # Keep only objects with
    exactly n cells
of_color(grid, color) -> Indices
              # get all indices of
    specified color
to_object(patch, grid) -> Object
              # convert patch to object
    with colors
shift_to_origin(obj) -> Patch
              # move object to origin
    (0,0)
to_indices(obj) -> Indices
              # extract just the (i,j)
    positions
as_indices(grid) -> Indices
              # all non-background cell
    positions
```

Modifications:
```python
fill(grid, color, patch) -> Grid
              # color cells in
    patch/indices
paint_onto_grid(grid, obj) -> Grid
              # draw object onto grid
replace(grid, old, new) -> Grid
              # swap all instances of
    old color with new
switch(grid, a, b) -> Grid
              # swap two colors
move_object(grid, obj, offset) -> Grid
              # move object by offset
shift_by_vector(patch, (di, dj)) ->
    Patch         # translate patch by
    offset
erase_patch(grid, patch) -> Grid
              # remove object (fill
    with background)
```

Spatial Queries:
```python
upper_left_corner(obj) -> IntegerTuple
              # upper-left corner
upper_right_corner(obj) -> IntegerTuple
              # upper-right corner
lower_left_corner(obj) -> IntegerTuple
              # lower-left corner
lower_right_corner(obj) -> IntegerTuple
              # lower-right corner
center(patch) -> IntegerTuple
              # center point of patch
corner_indices(patch) -> Indices
              # all 4 corner positions
position(a, b) -> IntegerTuple
              # relative position
    between patches
box(patch) -> Indices
              # outline of bounding box
inbox(patch) -> Indices
              # interior outline of box
outbox(patch) -> Indices
              # exterior outline of box
bounding_box_indices(patch) -> Indices
              # all indices in bounding
    box
bounding_box_delta(patch) -> Indices
              # bounding box minus the
    patch
vertical_line(loc) -> Indices
              # vertical line through
    location
horizontal_line(loc) -> Indices
              # horizontal line through
    location
shoot(start, direction) -> Indices
              # ray from point in
    direction
line_between(a, b) -> Indices
              # line between two points
```

Properties:
```python
get_height(grid) -> Integer
              # number of rows
get_width(grid) -> Integer
              # number of columns
get_shape(grid/obj) -> IntegerTuple
              # (height, width) tuple
size(container) -> Integer
              # number of elements
palette(grid) -> FrozenSet[Integer]
              # set of colors used
most_common_color(grid) -> Integer
              # most frequent color
    (background)
least_common_color(grid) -> Integer
              # least frequent color
get_color(obj) -> Integer
              # color of single-color
    object
color_at_location(grid, (i, j)) ->
    Integer/None   # value at location
occurrences(grid, obj) -> Indices
              # locations where obj
    appears
remove_solid_color_strips_from_grid(grid)
    -> Grid # remove uniform
```

```
    rows/columns
solid_color_strips_in_grid(grid) ->
    FrozenSet[Object] # get uniform
    rows/columns
horizontal_periodicity(obj) -> Integer
              # horizontal period
vertical_periodicity(obj) -> Integer
              # vertical period
move_until_touching(src, dst) ->
    IntegerTuple     # offset to move
    src next to dst
```

Set Operations:
```python
union(a, b) -> Container
              # union of containers
intersection(a, b) -> FrozenSet
              # intersection of sets
difference(a, b) -> FrozenSet
              # a - b (set difference)
flatten(containers) -> Container
              # flatten container of
    containers
remove_duplicates(tuple) -> Tuple
              # remove duplicates
keep_if_condition(container, f) ->
    Container      # filter by predicate
keep_if_condition_and_flatten(container,
    f) -> FrozenSet # filter and merge
contains(val, set) -> Boolean
              # membership test (val in
    set)
initset(value) -> FrozenSet
              # create singleton
    frozenset
```

Aggregation:
```python
argmax(container, f) -> Any
              # item with maximum
    f(item)
argmin(container, f) -> Any
              # item with minimum
    f(item)
valmax(container, f) -> Integer
              # maximum value of f over
    container
valmin(container, f) -> Integer
              # minimum value of f over
    container
maximum(container) -> Integer
              # max element
minimum(container) -> Integer
              # min element
most_common(container) -> Any
              # most frequent element
least_common(container) -> Any
              # least frequent element
sort(container, key) -> Tuple
              # sort by key function
```

Arithmetic:
```python
add(a, b) -> Numerical
              # addition (works on ints
    and tuples)
subtract(a, b) -> Numerical
              # subtraction
multiply(a, b) -> Numerical
              # multiplication
divide(a, b) -> Numerical
              # floor division
negate(n) -> Numerical
              # negation
double(n) -> Numerical
              # multiply by 2
halve(n) -> Numerical
              # divide by 2
increment(x) -> Numerical
              # add 1
decrement(x) -> Numerical
              # subtract 1
sign(x) -> Numerical
              # -1, 0, or 1 based on
    sign
to_vertical_vec(i) -> IntegerTuple
              # (i, 0) - vertical vector
to_horizontal_vec(j) -> IntegerTuple
              # (0, j) - horizontal
    vector
```

Constructors:
```python
create_grid(color, (height, width)) ->
    Grid       # Create blank grid
as_tuple(a, b, ...) -> IntegerTuple
              # create tuple from
    elements
repeat(item, n) -> Tuple
              # tuple with item
    repeated n times
```

Boolean:
```python
is_equal(a, b) -> Boolean
              # a == b
logical_and(a, b) -> Boolean
              # a and b
logical_or(a, b) -> Boolean
              # a or b
logical_not(b) -> Boolean
              # not b
contains(val, set) -> Boolean
              # val in set
is_positive(n) -> Boolean
              # n > 0
is_even(n) -> Boolean
              # n % 2 == 0
greater_than(a, b) -> Boolean
              # a > b
```

```
**Advanced:**
```python
cellwise(a, b, fallback) -> Grid
              # compare grids
    cell-by-cell
```

Constants:
```python
# Colors: ZERO=0, ONE=1, TWO=2,
    THREE=3, FOUR=4, FIVE=5, SIX=6,
    SEVEN=7, EIGHT=8, NINE=9
# Directions: UP=(-1,0), DOWN=(1,0),
    LEFT=(0,-1), RIGHT=(0,1)
# Diagonals: UNITY=(1,1),
    NEG_UNITY=(-1,-1), UP_RIGHT=(-1,1),
    DOWN_LEFT=(1,-1)
# Special: ORIGIN=(0,0), T=True, F=False
```

Control Flow Allowed
- Single-level `for` loops (no nesting)
- `if/else` conditionals
- List comprehensions

Requirements
- Function signature: `def solve(I):`
- Input `I` is tuple of tuples of ints
    (Grid)
- Return same format
- Use ONLY DSL primitives listed above
- Add brief comments for clarity
- Use functional programming patterns
    when appropriate (compose, chain,
    combine_two_function_results,
    fix_first_argument/fix_last_argument,
    transform_and_flatten)
- You can adapt patterns from similar
    programs but adjust them to match
    the pattern description

Output Format
```python
def solve(I):
    # [comment explaining step]
    x1 = dsl_function(I)

    # [comment]
    x2 = another_function(x1, args)

    # [final step]
    return O # O is the output grid

Below are 2 training examples follwed
    by the test example(s) you have to
    generalize to, for each example,
    the input grid is shown first,
    followed by the output grid.

Example 1:
Input:

ASCII representation:
0|0|0|0|0|0|0|0|0|0|0
1|0|0|0|0|0|0|0|0|0|2
0|0|0|0|0|0|0|0|0|0|0
0|0|0|0|0|0|0|0|0|0|0
0|0|0|0|0|0|0|0|0|0|0

Output:

ASCII representation:
0|0|0|0|0|0|0|0|0|0|0
1|1|1|1|1|5|2|2|2|2|2
0|0|0|0|0|0|0|0|0|0|0
0|0|0|0|0|0|0|0|0|0|0
0|0|0|0|0|0|0|0|0|0|0

Example 2:
Input:

ASCII representation:
0|0|0|0|0|0|0|0|0|0|0
0|0|0|0|0|0|0|0|0|0|0
0|0|0|0|0|0|0|0|0|0|0
3|0|0|0|0|0|0|0|0|0|7
0|0|0|0|0|0|0|0|0|0|0

Output:

ASCII representation:
0|0|0|0|0|0|0|0|0|0|0
0|0|0|0|0|0|0|0|0|0|0
0|0|0|0|0|0|0|0|0|0|0
3|3|3|3|3|5|7|7|7|7|7
0|0|0|0|0|0|0|0|0|0|0


============================================================
TEST EXAMPLES (to solve)
============================================================
Below are 1 test example(s) you need to
    solve:
For each test example, only the input
    grid is provided. You must
    determine the output.

Test Example 1:
Input:

ASCII representation:
0|0|0|0|0|0|0|0|0|0|0
4|0|0|0|0|0|0|0|0|0|8
0|0|0|0|0|0|0|0|0|0|0
0|0|0|0|0|0|0|0|0|0|0
6|0|0|0|0|0|0|0|0|0|9

Output: [TO BE DETERMINED]

Solution:
```python
def solve(I):
    x1 = lefthalf(I)
    x2 = righthalf(I)
    x3 = objects(x2, T, F, T)
```
```
```

```
    x4 = objects(x1, T, F, T)
    x5 = compose(hfrontier, center)
    x6 = fork(recolor, color, x5)
    x7 = mapply(x6, x4)
    x8 = paint(x1, x7)
    x9 = mapply(x6, x3)
    x10 = paint(I, x9)
    x11 = objects(x8, T, F, T)
    x12 = apply(urcorner, x11)
    x13 = shift(x12, RIGHT)
    x14 = merge(x11)
    x15 = paint(x10, x14)
    O = fill(x15, FIVE, x13)
    return O
```

Task 2:

Below are 3 training examples follwed
    by the test example(s) you have to
    generalize to, for each example,
    the input grid is shown first,
    followed by the output grid.

Example 1:
Input:

ASCII representation:
0|0|0|0|0|0|0|0|0|0|0|0|0
0|0|1|0|1|0|0|1|1|0|1|0|0
0|0|1|0|0|0|0|0|0|0|0|0|0
0|0|0|0|0|0|0|0|0|0|1|0|0
0|0|0|0|0|0|0|0|0|0|0|0|0
0|0|0|0|0|0|0|0|0|0|1|0|0
0|0|1|0|0|0|0|0|0|0|1|0|0
0|0|1|1|0|0|1|1|0|1|1|0|0
0|0|0|0|0|0|0|0|0|0|0|0|0

Output:

ASCII representation:
0|0|0|0|0|0|0|0|0|0|0|0|0
0|0|1|2|1|2|2|1|1|2|1|0|0
0|0|1|0|0|0|0|0|0|0|2|0|0
0|0|2|0|0|0|0|0|0|0|1|0|0
0|0|2|0|0|0|0|0|0|0|2|0|0
0|0|2|0|0|0|0|0|0|0|1|0|0
0|0|1|0|0|0|0|0|0|0|1|0|0
0|0|1|1|2|2|1|1|2|1|1|0|0
0|0|0|0|0|0|0|0|0|0|0|0|0

Example 2:
Input:

ASCII representation:
0|0|0|0|0|0|0|0|0|0|0|0|0
0|0|0|0|0|0|0|0|0|0|0|0|0
0|0|1|1|1|0|0|1|1|0|0|0|0
0|0|1|0|0|0|0|0|1|0|0|0|0
0|0|0|0|1|0|0|0|0|0|0|0|0
0|0|1|0|1|0|0|0|1|0|0|0|0
0|0|1|0|0|0|0|0|1|0|0|0|0
0|0|0|0|1|0|0|0|1|0|0|0|0
0|0|1|1|1|1|0|1|0|0|0|0|0
0|0|0|0|0|0|0|0|0|0|0|0|0
0|0|0|0|0|0|0|0|0|0|0|0|0

Output:

ASCII representation:
0|0|0|0|0|0|0|0|0|0|0|0|0
0|0|0|0|0|0|0|0|0|0|0|0|0
0|0|1|1|2|2|1|1|0|0|0|0|0
0|0|1|0|2|0|0|0|1|0|0|0|0
0|0|2|0|1|0|0|0|2|0|0|0|0
0|0|1|0|1|0|0|0|1|0|0|0|0
0|0|1|0|2|0|0|0|1|0|0|0|0
0|0|2|0|1|0|0|0|1|0|0|0|0
0|0|1|1|1|1|2|1|2|0|0|0|0
0|0|0|0|0|0|0|0|0|0|0|0|0
0|0|0|0|0|0|0|0|0|0|0|0|0

Example 3:
Input:

ASCII representation:
0|0|0|0|0|0|0|0|0|0|0|0|0
0|0|0|0|0|0|0|0|0|0|0|0|0
0|0|0|0|0|0|0|0|0|0|0|0|0
0|0|1|1|0|1|1|0|1|1|1|0|0
0|0|1|0|0|0|0|0|0|0|1|0|0
0|0|0|0|0|0|0|0|0|0|0|0|0
0|0|1|0|0|0|0|0|0|0|1|0|0
0|0|1|1|0|1|0|1|1|0|0|0|0
0|0|1|0|0|0|0|0|0|0|1|0|0
0|0|0|0|0|0|0|0|0|0|1|0|0
0|0|1|1|0|1|1|0|0|1|1|0|0
0|0|0|0|0|0|0|0|0|0|0|0|0
0|0|0|0|0|0|0|0|0|0|0|0|0

Output:

ASCII representation:
0|0|0|0|0|0|0|0|0|0|0|0|0
0|0|0|0|0|0|0|0|0|0|0|0|0
0|0|0|0|0|0|0|0|0|0|0|0|0
0|0|1|1|2|1|1|2|1|1|1|0|0
0|0|1|0|0|0|0|0|0|0|1|0|0
0|0|2|0|0|0|0|0|0|0|2|0|0
0|0|1|0|0|0|0|0|0|0|1|0|0
0|0|1|1|2|1|2|1|1|2|2|0|0
0|0|1|0|0|0|0|0|0|0|1|0|0
0|0|2|0|0|0|0|0|0|0|1|0|0
0|0|1|1|2|1|1|2|2|1|1|0|0
0|0|0|0|0|0|0|0|0|0|0|0|0
0|0|0|0|0|0|0|0|0|0|0|0|0

==========
TEST EXAMPLES (to solve)
==========
Below are 1 test example(s) you need to
    solve:
For each test example, only the input
    grid is provided. You must

```
    determine the output.

Test Example 1:
Input:

ASCII representation:
0|0|0|0|0|0|0|0|0|0|0|0|0
0|0|0|0|0|0|0|0|0|0|0|0|0
0|0|1|0|1|1|0|1|0|1|1|0|0
0|0|1|0|0|0|0|0|0|0|1|0|0
0|0|0|0|0|0|0|0|0|0|0|0|0
0|0|0|0|0|0|0|0|0|0|0|0|0
0|0|1|0|0|0|0|0|0|0|1|0|0
0|0|1|0|1|0|1|0|0|1|1|0|0
0|0|0|0|0|0|0|0|0|0|0|0|0
0|0|1|0|0|0|0|0|0|0|1|0|0
0|0|1|0|1|1|0|1|0|1|1|0|0
0|0|0|0|0|0|0|0|0|0|0|0|0
0|0|0|0|0|0|0|0|0|0|0|0|0

Output: [TO BE DETERMINED]

Solution:
```python
def solve(I):
    x1 = ofcolor(I, ONE)
    x2 = box(x1)
    x3 = fill(I, TWO, x2)
    x4 = subgrid(x1, x3)
    x5 = ofcolor(x4, ONE)
    x6 = mapply(vfrontier, x5)
    x7 = mapply(hfrontier, x5)
    x8 = size(x6)
    x9 = size(x7)
    x10 = greater(x8, x9)
    x11 = branch(x10, x7, x6)
    x12 = fill(x4, TWO, x11)
    x13 = ofcolor(x12, TWO)
    x14 = ulcorner(x1)
    x15 = shift(x13, x14)
    O = underfill(I, TWO, x15)
    return O
```

Task 3:

Below are 2 training examples follwed
    by the test example(s) you have to
    generalize to:
 for each example, the input grid is
    shown first, followed by the output
    grid.
.
Example 1:
Input:

ASCII representation:
8|0|0|0|0|0|8|8|8|8|8|8|0|8|8|8|0|8|8|0|8|8|8
0|0|8|8|8|0|0|0|0|0|0|8|0|0|0|8|0|8|0|0|8|0|8
8|8|8|0|8|0|8|8|8|8|0|8|8|8|0|8|0|8|8|8|8|0|8
8|0|0|0|8|0|8|0|0|8|0|0|0|8|0|8|0|0|0|0|0|0|8
8|0|8|8|8|0|8|8|0|8|0|8|8|8|0|8|8|0|8|8|8|8|8
8|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0
8|0|8|8|8|0|8|8|8|8|8|8|0|8|8|8|0|8|8|0|8|8|8
8|0|8|8|8|0|8|8|8|0|8|8|8|0|8|8|8|0|8|8|8|8|8
8|0|8|8|8|0|8|8|8|8|8|8|0|8|8|8|0|8
```

```
For each test example, only the input
    grid is provided. You must
    determine the output.

Test Example 1:
Input:

ASCII representation:
8|8|0|8|0|0|8|0|0|0|0|0|0|0|0
0|8|0|8|8|8|8|4|8|8|8|8|8|8|8
0|8|0|0|0|0|4|3|8|0|0|0|0|0|8
0|8|8|8|8|8|8|4|8|8|8|0|8|8|8
0|0|0|0|0|0|8|0|0|0|8|0|8|0|0
8|8|8|8|8|0|8|8|8|0|8|0|8|0|8
0|0|0|0|8|0|0|0|8|0|8|0|8|0|8
8|8|8|0|8|8|8|0|8|0|8|0|8|8|8
0|0|8|0|0|0|8|0|8|0|8|0|0|0|0
8|0|8|8|8|0|8|8|8|0|8|8|8|0|8
8|0|0|0|8|0|0|0|0|0|0|0|8|0|8
8|8|8|0|8|0|8|8|8|8|8|8|8|0|8
0|0|8|0|8|0|8|0|0|0|0|0|0|0|8
8|0|8|8|8|0|8|0|8|8|8|8|8|8|8
8|0|0|0|0|0|8|0|8|0|0|0|0|0|0

Output: [TO BE DETERMINED]

Solution:
```python
def solve_b782dc8a(I):
    x1 = least_common_color(I)
    x2 = as_objects(I, True, False,
    False)
    x3 = of_color(I, x1)
    x4 = get_first(x3)
    x5 = direct_neighbors(x4)
    x6 = to_object(x5, I)
    x7 = most_common_color(x6)
    x8 = of_color(I, x7)
    x9 = color_filter(x2, COLOR_ZERO)
    x10 = fix_last_argument(adjacent,
    x8)
    x11 =
    keep_if_condition_and_flatten(x9,
    x10)
    x12 = to_indices(x11)
    x13 =
    fix_last_argument(manhattan_distance,
    x3)
    x14 = chain(is_even, x13, initset)
    x15 = keep_if_condition(x12, x14)
    x16 = difference(x12, x15)
    x17 = fill(I, x1, x15)
    O = fill(x17, x7, x16)
    return O
```

Generate the `solve(I)` function now.
```

B.4. Prompt 3b. Program Evolution

The program evolution prompt repairs a failed program by conditioning on the execution error while keeping the hypothesis as a semantic anchor.

Prompt used in experiments

```
You are an expert Python programmer
    fixing code.

## Repair Task

The following code was generated to
    solve the task, but it failed on
    the training examples.

### Generated Code

python{code}

### Error / Failure

{error}

### Training Examples (for reference)

Please fix the code. Return the full
    corrected `solve(I)` function.
```

.